Moneta: A High-performance Storage Array Architecture for Next-generation, Non-volatile Memories

Adrian M. Caulfield Arup De Joel Coburn Todor I. Mollov
Rajesh K. Gupta Steven Swanson

Department of Computer Science and Engineering
University of California, San Diego
{acaulfie,arde,jdcoburn,rgupta,swanson}@cs.ucsd.edu, tmollov@ucsd.edu

***Abstract*—Emerging non-volatile memory technologies such as phase change memory (PCM) promise to increase storage system performance by a wide margin relative to both conventional disks and flash-based SSDs. Realizing this potential will require significant changes to the way systems interact with storage devices as well as a rethinking of the storage devices themselves. This paper describes the architecture of a prototype PCIe-attached storage array built from emulated PCM storage called *Moneta*. Moneta provides a carefully designed hardware/software interface that makes issuing and completing accesses atomic. The atomic management interface, combined with hardware scheduling optimizations, and an optimized storage stack increases performance for small, random accesses by 18x and reduces software overheads by 60%. Moneta array sustain 2.8 GB/s for sequential transfers and 541K random 4 KB IO operations per second (8x higher than a state-of-the-art flash-based SSD). Moneta can perform a 512-byte write in 9 us (5.6x faster than the SSD). Moneta provides a harmonic mean speedup of 2.1x and a maximum speed up of 9x across a range of file system, paging, and database workloads. We also explore trade-offs in Moneta's architecture between performance, power, memory organization, and memory latency.**

***Keywords*-software IO optimizations; non-volatile memories; phase change memories; storage systems**

I. Introduction

For many years, the performance of persistent storage (i.e., disks) has lagged far behind that of microprocessors. Since 1970, microprocessor performance grew by roughly 200,000×. During the same period, disk access latency has fallen by only 9× while bandwidth has risen by only 163× [1, 2].

The emergence of non-volatile, solid-state memories (such as NAND flash and phase-change memories, among others) has signaled the beginning of the end for painfully slow non-volatile storage. These technologies will potentially reduce latency and increase bandwidth for non-volatile storage by many orders of magnitude, but fully harnessing their performance will require overcoming the legacy of disk-based storage systems.

This legacy takes the form of numerous hardware and software design decisions that assume that storage is slow. The hardware interfaces that connect individual disks to computer systems are sluggish ($\sim$300 MB/s for SATA II and SAS, 600 MB/s for SATA 600) and connect to the slower "south bridge" portion of the CPU chip set [3]. RAID controllers connect via high-bandwidth PCIe, but the low-performance, general-purpose microprocessors they use to schedule IO requests limit their throughput and add latency [4].

Software also limits IO performance. Overheads in the operating system's IO stack are large enough that, for solid-state storage technologies, they can exceed the hardware access time. Since it takes $\sim$20,000 instructions to issue and complete a 4 KB IO request under standard Linux, the computational overhead of performing hundreds of thousands of IO requests per second can limit both IO and application performance.

This paper describes a prototype high-performance storage array, called Moneta[1], designed for next-generation non-volatile memories, such as phase-change memory, that offer near-DRAM performance. Moneta allows us to explore the architecture of the storage array, the impact of software overheads on performance, the effects of non-volatile technology parameters on bandwidth and latency, and the ultimate benefit to applications. We have implemented Moneta using a PCIe-attached array of FPGAs and DRAM. The FPGAs implement a scheduler and a distributed set of configurable memory controllers. The controllers allow us to emulate fast non-volatile memories by accurately modeling memory technology parameters such as device read and write times, array geometry, and internal buffering.

Achieving high performance in Moneta requires simultaneously optimizing its hardware and software components. We characterize the overheads in the existing Linux IO stack in detail, and show that a redesigned IO stack combined with an optimized hardware/software interface reduces IO latency by nearly 2× and increases bandwidth by up to 18×. Tuning the Moneta hardware improve bandwidth by an additional 75% for some workloads.

We present two findings on the impact of non-volatile mem-

[1] "Moneta" is the Latin name for the goddess of memory.

ory performance and organization on Moneta's performance and energy efficiency. First, some optimizations that improve PCM's performance and energy efficiency as a main memory technology do not apply in storage applications because of different usage patterns and requirements. Second, for 4 KB accesses, Moneta provides enough internal parallelism to completely hide memory access times of up to 1 μs, suggesting that memory designers could safely trade off performance for density in memory devices targeted at storage applications.

Results for a range of IO benchmarks demonstrate that Moneta outperforms existing storage technologies by a wide margin. Moneta can sustain up to 2.2 GB/s on random 4 KB accesses, compared to 250 MB/s for a state-of-the-art flash-based SSD. It can also sustain over 1.1 M 512 byte random IO operations per second. While Moneta is nearly 10× faster than the flash drive, software overhead beyond the IO stack (e.g., in the file system and in application) limit application-level speedups: Compared to the same flash drive, Moneta speeds up applications by a harmonic mean of just 2.1×, demonstrating that further work is necessary to fully realize Moneta's potential at the application level.

The remainder of the paper is organized as follows. Section II describes the Moneta array. Sections III, IV, and V analyze its performance and describe software and hardware optimizations. Section VI compares Moneta to existing storage technologies, and Section VII describes related work. Finally, Section VIII concludes.

II. The Moneta prototype

This section briefly discusses the memory technologies that Moneta targets and then describes the baseline Moneta architecture and the hardware system we use to emulate it.

A. Non-volatile memories

Several fast, non-volatile memory technologies are vying to replace or supplement flash as a solid-state storage system building block. These include phase-change memories (PCM), as well as spin-torque transfer memories [5], memristor [6], race-track [7], and carbon nanotube-based storage devices [8]. PCM is the most promising of these upcoming memory technologies.

In this work we model PCM as the storage technology for Moneta, but the basic design of Moneta does not rely on the particular characteristics of PCM. Moneta assumes the memory technology has performance (latency and bandwidth) close to that of DRAM and that it presents a DRAM-like interface.

Phase change memory (PCM) stores data as the crystalline state of a chalcogenide metal layer [9], and recent work [10, 11] has demonstrated that PCM may become a viable main memory technology as DRAM's scaling begins to falter. The analysis in [10] provides a good characterization of PCM's performance and power consumption.

Despite this promise, PCM does suffer from some reliability concerns: A PCM bit has a typical lifetime of 10^8 program cycles, so it requires management to ensure reasonable device lifetime. Several PCM wear-leveling schemes exist [10, 12–15]. Moneta uses the start-gap scheme in [15] which provides wear-leveling with less than 1% overhead.

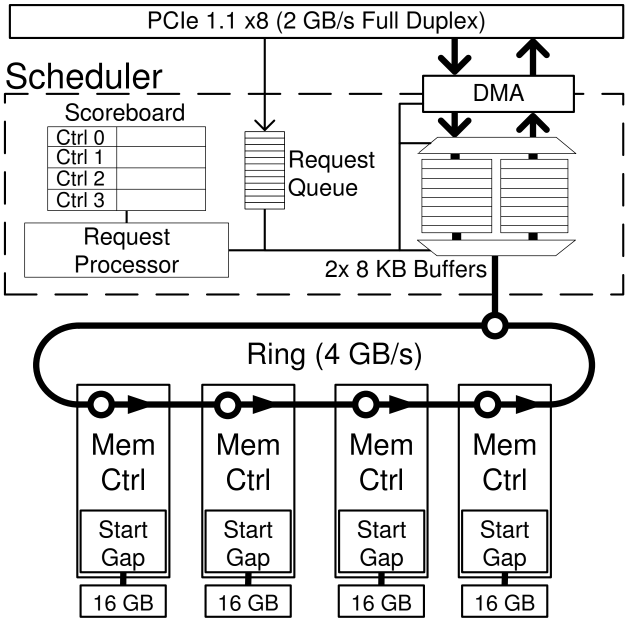

Fig. 1. **The Moneta system** Moneta's four PCM memory controllers connect to the scheduler via a 4 GB/s ring. A 2 GB/s full duplex PCIe link connects the scheduler and DMA engine to the host. The scheduler manages two 8 KB buffers as it processes IO requests in FIFO order. A scoreboard tracks the state of the memory controllers.

B. Moneta array architecture

Figure 1 shows the architecture of the Moneta storage array. Moneta's architecture provides low-latency access to a large amount of non-volatile memory spread across four memory controllers. A scheduler manages the entire array by coordinating data transfers over the PCIe interface and the ring-based network that connects the independent memory controllers to a set of input/output queues. Moneta attaches to the computer system via an eight-lane PCIe 1.1 interface that provides a 2 GB/s full-duplex connection (4 GB/s total).

The Moneta scheduler

The scheduler orchestrates Moneta's operation. It contains a DMA controller and a Programmed IO (PIO) interface to communicate with the host machine, a set of internal buffers for incoming and outgoing data, several state machines, and an interface to the 128-bit token-ring network. The Moneta scheduler stripes internal storage addresses across the memory controllers to extract parallelism from large requests. The baseline stripe size is 8 KB.

Requests arrive on the PCIe interface as PIO writes from the software driver. Each request comprises three 64-bit words that contain a sector address, a DMA memory address (in host memory), a 32-bit transfer length, and a collection of control bits that includes a tag number and an op code (read or write). Sectors are 512 bytes. The tag is unique across outstanding requests and allows for multiple in-flight requests, similar to SATA's Native Command Queuing [16].

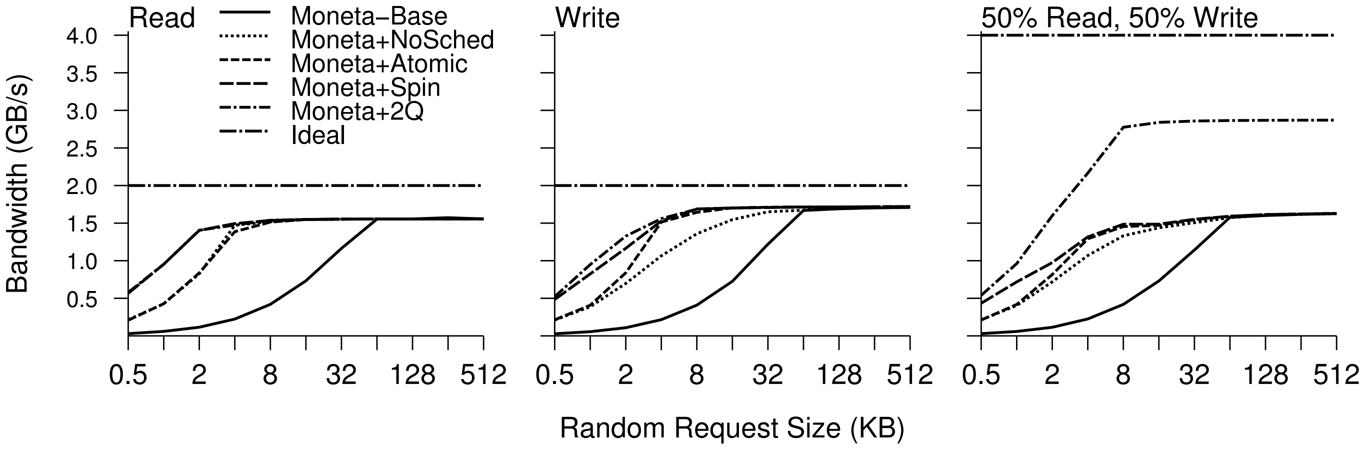

Fig. 2. **Moneta bandwidth measurements** The PCIe link bandwidth (labeled "ideal") limits performance for large accesses, but for small accesses software overheads are the bottleneck for the baseline Moneta design (the solid black line). The other lines measure results for optimized version described in Section IV and V.

The scheduler places requests into a FIFO queue as they arrive and processes them in order. Depending on the request's size and alignment, the scheduler breaks it into one or more transfers of up to 8 KB. It then allocates a buffer for each transfer from two 8 KB buffers in the scheduler. If buffer space is not available, the scheduler stalls until another transfer completes.

For write transfers, the scheduler issues a DMA command to transfer data from the host's memory into its buffer. When the DMA transfer completes, the scheduler checks an internal scoreboard to determine whether the target memory controller has space to receive the data, and waits until sufficient space is available. Once the transfer completes, the scheduler's buffer is available for another transfer. The steps for a read transfer are similar except that the steps are reversed.

Once the scheduler completes all transfers for a request, it raises an interrupt and sets a tag status bit. The operating system receives the interrupt and completes the request by reading and then clearing the status bit using PIO operations.

The DMA controller manages Moneta's 2 GB/s full-duplex (4 GB/s total) channel to the host system. The DMA interleaves portions of bulk data transfers from Moneta to the host's memory (read requests) with DMA requests that retrieve data from host's memory controller (write requests). This results in good utilization of the bandwidth between the host and Moneta because bulk data transfers can occur in both directions simultaneously.

Memory controllers

Each of Moneta's four memory controllers manages an independent bank of non-volatile storage. The controllers connect to the scheduler via the ring and provide a pair of 8 KB queues to buffer incoming and outgoing data. The memory array at each controller comprises four DIMM-like memory modules that present a 72 bit (64 + 8 for ECC) interface.

Like DRAM DIMMs, the memory modules perform accesses in parallel across multiple chips. Each DIMM contains four internal banks and two ranks, each with an internal row buffer. The banks and their row buffers are 8 KB wide, and the DIMM reads an entire row from the memory array into the row buffer. Once that data is in the row buffer, the memory controller can access it at 250 MHz DDR (500M transfers per second), so the peak bandwidth for a single controller is 4 GB/s.

The memory controller implements the start-gap wear-leveling and address randomization scheme [15] to evenly distribute wear across the memory it manages.

C. Implementing the Moneta prototype

We have implemented a Moneta prototype using the BEE3 FPGA prototyping system designed by Microsoft Research for use in the RAMP project [17]. The BEE3 system holds 64 GB of 667 MHz DDR2 DRAM under the control of four Xilinx Virtex 5 FPGAs, and it provides a PCIe link to the host system. The Moneta design runs at 250 MHz, and we use that clock speed for all of our results.

Moneta's architecture maps cleanly onto the BEE3. Each of the four FPGAs implement a memory controller, while one also implements the Moneta scheduler and the PCIe interface. The Moneta ring network runs over the FPGA-to-FGPA links that the BEE3 provides.

The design is very configurable. It can vary the effective number of memory controllers without reducing the amount of memory available, and it supports configurable buffer sizes in the scheduler and memory controllers.

Moneta memory controllers emulate PCM devices on top of DRAM using a modified version of the Xilinx Memory Interface Generator DDR2 controller. It adds latency between the read address strobe and column address strobe commands during reads and extends the precharge latency after a write. The controller can vary the apparent latencies for accesses to memory from 4 ns to over 500 μs. We use the values from [10] (48 ns and 150 ns for array reads and writes, respectively) to model PCM in this work, unless otherwise stated.

The width of memory arrays and the corresponding row buffers are important factors in the performance and energy efficiency of PCM memory [10]. The memory controller can vary the effective width of the arrays by defining a virtual row size and inserting latencies to model opening and closing rows of that size. The baseline configuration uses 8 KB rows.

III. Baseline Moneta performance

We use three metrics to characterize Moneta's performance. The first, shown in Figure 2, is a curve of sustained bandwidth for accesses to randomly selected locations over a range of access sizes. The second measures the average latency for a 4 KB access to a random location on the device. The final metric is the number of random access 4 KB IO operations per second (IOPS) the device can perform with multiple threads.

Our test system is a two-socket, Core i7 Quad (a total of 8 cores) machine running at 2.26 GHz with 8 GB of physical DRAM and two 8 MB L2 caches. All measurements in this section are for a bare block device without a file system. We access Moneta via a low-level block driver and the standard Linux IO stack configured to use the No-op scheduler, since it provided the best performance. Each benchmark reports results for reads, writes, and a 50/50 combination of the two. We use XDD [18], a configurable IO performance benchmark for these measurements.

Figure 2 shows bandwidth curves for the baseline Moneta array and the optimized versions we discuss in future sections. The "ideal" curve shows the bandwidth of the 2 GB/s PCIe connection.

The baseline Moneta design delivers good performance, but there is room for improvement for accesses smaller than 64KB. For these, long access latency limits bandwidth, and much of this latency is due to software. Figure 3 breaks down the latency for 4 KB read and write accesses into 14 components. The data show that software accounts for 13 μs of the total 21 μs latency, or 62%. For comparison, the software overheads for issuing a request to the SSD- and disk-based RAID arrays described in Section VI are just 17% and 1%, respectively. Although these data do not include a file system, other measurements show file system overhead adds an additional 4 to 5 μs of delay to each request.

These software overheads limit the number of IOPS that a single processor can issue to 47K, so it would take 10.5 processors to achieve Moneta's theoretical peak of 500K 4 KB read IOPS. As a result, IO-intensive applications running on Moneta with fewer than 11 processors will become CPU bound before they saturate the PCIe connection to Moneta.

These overheads demonstrate that software costs are the main limiter on Moneta performance. The next section focuses on minimizing these overheads.

IV. Minimizing software costs

Many of the software overheads in Figure 3 result from legacy optimizations that improve performance for slow, conventional non-volatile storage (i.e. disks). Moneta is faster than legacy storage systems, so many of those optimizations are no longer profitable. We begin by removing unprofitable optimizations, and then move on to optimizing other costs that limit Moneta's performance but did not warrant attention for slow storage. Figures 2 and 3 track the improvement of each optimization in terms of bandwidth and latency.

A. IO scheduler

Linux IO schedulers sort and merge requests to reduce latencies and provide fair access to IO resources under load. Reordering requests can provide significant reductions in latency for devices with non-uniform access latencies such as disks, but for Moneta it just adds software overheads. Even the no-op scheduler, which simply passes requests directly to the driver without any scheduling, adds 2 μs to each request. This cost arises from context switches between the thread that requested the operation and the scheduler thread that actually performs the operation.

Moneta+NoSched bypasses the scheduler completely. It uses the thread that entered the kernel from user space to issue the request without a context switch. Removing the scheduler provides several ancillary benefits, as well. The no-op scheduler is single-threaded so it removes any parallelism in request handling. Under Moneta+NoSched each request has a dedicated thread and those threads can issue and complete requests in parallel. Moneta+NoSched reduces per-request latency by 2 μs and increases peak bandwidth by 4× for 4 KB requests.

B. Issuing and completing IO requests

Moneta+NoSched allows some parallelism, but threads still contend for access to two locks that protect Moneta's hardware interface and shared data structures, limiting effective parallelism in the driver. The first lock protects the hardware interface during multi-word reads and writes to Moneta's PIO interface. The second lock protects shared data structures in the software.

To safely remove the lock that protects Moneta's hardware/software interface, we modify the interface so that issuing and completing a request requires a single, atomic PIO write or read. The baseline interface requires several PIO writes to issue a request. Moneta+Atomic reduces this to one by removing bits from the internal address and length fields to align them to 512 byte boundaries and completely removing the DMA address field. These changes allow a request to fit into 64 bits – 8 for the tag, 8 for the command, 16 for the length, and 32 for the internal address.

To specify the target address for DMA transfers, the Moneta+Atomic driver pre-allocates a DMA buffer for each tag and writes the host DRAM address of each buffer into a register in the hardware. The tag uniquely identifies which DMA buffer to use for each request.

A second change to Moneta's hardware/software interface allows multiple threads to process interrupts in parallel. When a request completes and Moneta raises an interrupt, the driver checks the status of all outstanding operations by reading the tag status register. In Moneta+NoSched the tag status register

Label	Description	Baseline latency (μs)	
		Write	Read
OS/User	OS and userspace overhead	1.98	1.95
OS/User	Linux block queue and no-op scheduler	2.51	3.74
Schedule	Get request from queue and assign tag	0.44	0.51
Copy	Data copy into DMA buffer	0.24/KB	-
Issue	PIO command writes to Moneta	1.18	1.15
DMA	DMA from host to Moneta buffer	0.93/KB	-
Ring	Data from Moneta buffer to mem ctrl	0.28/KB	-
PCM	4 KB PCM memory access	4.39	5.18
Ring	Data from mem ctrl to Moneta buffer	-	0.43/KB
DMA	DMA from Moneta buffer to host	-	0.65/KB
Wait	Thread sleep during hw	11.8	12.3
Interrupt	Driver interrupt handler	1.10	1.08
Copy	Data copy from DMA buffer	-	0.27/KB
OS/User	OS return and userspace overhead	1.98	1.95
Hardware total for 4 KB (accounting for overlap)		8.2	8.0
Software total for 4 KB (accounting for overlap)		13.3	12.2
File system additional overhead		5.8	4.2

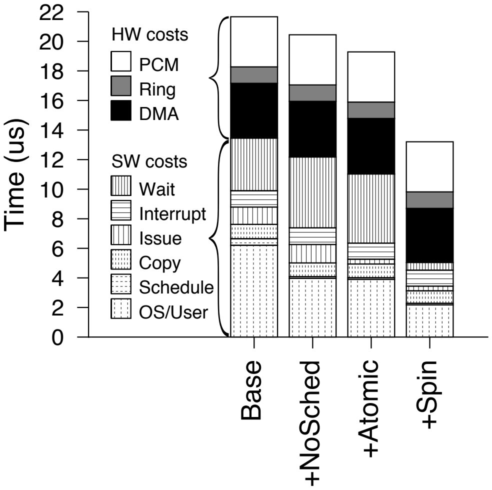

Fig. 3. **Latency savings from software optimizations** The table breaks down baseline Moneta latency by component. The figure shows that software optimizations reduce overall access latency by 9 μs between Moneta-Base and Moneta+Spin. Total latency is smaller than the sum of the components, due to overlap among components.

indicates whether each tag is busy or idle and the driver must read the register and then update it inside a critical section protected by a lock. In Moneta+Atomic the tag status bits indicate whether a request using that tag finished since the last read of the register, and reading the register clears it. Atomically reading and clearing the status register allows threads to service interrupts in parallel without the possibility of two threads trying to complete the same request.

The second lock protects shared data structures (e.g., the pool of available tags). To remove that lock, reimplemented the tag pool data as a lock-free data structure, and allocate other structures on a per-tag basis.

By removing all the locks from the software stack, Moneta+Atomic reduces latency by 1 μs over Moneta+NoSched and increases bandwidth by 460 MB/s for 4 KB writes. The disproportionate gain in bandwidth versus latency results from increased concurrency.

C. Avoiding interrupts

Responding to the interrupt that signals the completion of an IO request requires a context switch to wake up the thread that issued the request. This process adds latency and limits performance, especially for small requests. Allowing the thread to spin in a busy-loop rather than sleeping removes the context switch and the associated latency. Moneta+Spin implements this optimization.

Spinning reduces latency by 6 μs, but it increases per-request CPU utilization. It also means that the number of thread-contexts available bounds the number outstanding requests. Our data show that spinning only helps for write requests smaller than 4 KB, so the driver spins for those requests and sleeps for larger requests. Moneta+Spin improves bandwidth for 512-4096 byte requests by up to 250 MB/s.

D. Other overheads

The remaining overheads are from the system call/return and the copies to and from userspace. These provide protection for the kernel. Optimizing the system call interface is outside the scope of this paper, but removing copies to and from userspace by directing DMA transfers into userspace buffers is a well-known optimization in high-performance drivers. For Moneta, however, this optimization is not profitable, because it requires sending a physical address to the hardware as the target for the DMA transfer. This would make issuing requests atomically impossible. Our measurements show that this hurts performance more than removing the copy to or from userspace helps. One solution would be to extend the processor's ISA to support 128 bit atomic writes. This would allow an atomic write to include the full DMA target address.

V. Tuning the Moneta hardware

With Moneta's optimized hardware interface and IO stack in place, we shift our focus to four aspects of the Moneta hardware – increasing simultaneous read/write bandwidth, increasing fairness between large and small transfers, adapting to memory technologies with longer latencies than PCM, and power consumption.

A. Read/Write bandwidth

Figure 2 shows that Moneta+Spin nearly saturates the PCIe link for read- and write-only workloads. For the mixed workload, performance is similar, but performance should be better since the PCIe link is full duplex.

Moneta+Spin uses a single hardware queue for read and write requests which prevents it from fully utilizing the PCIe link. Moneta+2Q solves this problem by providing separate queues for reads and writes and processing the queues in

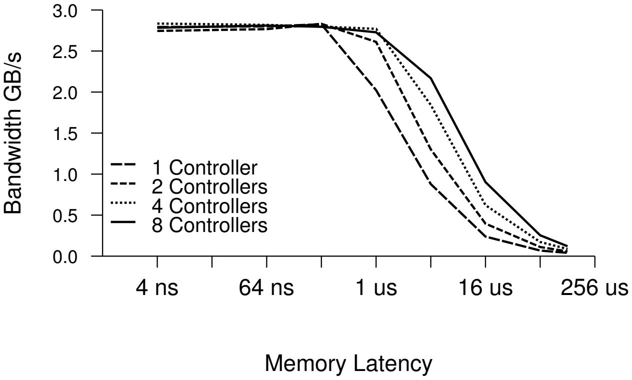

Fig. 4. **Managing long-latency non-volatile memories** Additional controllers allow Moneta to hide longer memory latencies without sacrificing bandwidth. The curves measure bandwidth for 4 KB read and write accesses.

round-robin order. When reads and writes are both present, half of the requests to the DMA engine will be reads and half will be writes. The DMA engine interleaves the requests to transfer data in both directions simultaneously.

Figure 2 includes performance for Moneta+2Q. It increases mixed read/write performance by 75% and consumes 71% of the total PCIe bandwidth.

The new scheduler adds some complexity to the design and requires a change to how we implement IO barriers. IO barriers prevent the storage system from moving requests across the barrier, and some applications (e.g. file systems) use barriers to enforce ordering. The baseline Moneta design enforces this constraint by processing requests in order. Moneta+2Q's scheduler keeps track of outstanding barriers and synchronizes the read and write queue as needed to implement them. Requests that enter either queue after a barrier will wait for the barrier to complete before issuing.

B. *Balancing bandwidth*

Real-world application access patterns are more complex and varied than the workloads we have tested so far. Multiple applications will run concurrently placing different demands on Moneta simultaneously. Ideally, Moneta would prevent one access or access pattern from unduly degrading performance for other accesses.

The scheduler in Moneta+2Q processes the requests in each queue in order, so large requests can significantly delay other requests in the same queue. For a mixture of large and small requests, both sizes perform the same number of *accesses* per second, but bandwidth will be much smaller for the smaller accesses. For instance, if four threads issuing 4 KB reads run in parallel with four threads issuing 512 KB reads, the 4 KB threads realize just 16 MB/s, even though a 4 KB thread running in isolation can achieve 170 MB/s. The 512 KB threads receive a total 1.5 GB/s.

To reduce this imbalance, we modified the scheduler to service requests in the queue in round-robin order. The scheduler allocates a single buffer to the request at the front of the queue and then places the remainder of the request back on the queue before moving on to the next request. Round-robin queuing improves performance for the 4 KB threads in our example by 12×, and reduces large request bandwidth by 190 MB/s. Aggregate bandwidth remains the same.

The modified scheduler implements barriers by requiring that all in-progress requests complete before inserting any operations that follow the barrier into the scheduling queues.

C. *Non-volatile memory latency*

It is difficult to predict the latencies that non-volatile memories will eventually achieve, and it is possible that latencies will be longer than those we have modeled.

Figure 4 explores the impact of memory latency on Moneta's performance. It plots Moneta+Balance performance with memory latencies between 4 ns to 128 μs and with 1 to 8 memory controllers. The data show that adding parallelism in the form of memory controllers can completely hide latencies of up to 1 μs and that 4 μs memories would only degrade performance by 20% with eight controllers. The data also show that at 4 μs, doubling the number of controllers increases bandwidth by approximately 400 MB/s.

These results suggest that, for bandwidth-centric applications, it makes sense to optimize non-volatile memories for parallelism and interface bandwidth rather than latency. Furthermore, Moneta's memory controller only accesses a single bank of memory at a time. A more advanced controller could leverage inter-bank parallelism to further hide latency. Flash-based SSDs provide a useful case study: FusionIO's ioDrive, a state-of-the-art SSD, has many independent memory controllers to hide the long latencies of flash memory. In Section VI we show that while a FusionIO drive can match Moneta's performance for large transfers, its latency is much higher.

D. *Moneta power consumption*

Non-volatile storage reduces power consumption dramatically compared to hard drives. To understand Moneta's power requirements we use the power model in Table I. For the PCM array, we augment the power model in [10] to account for a power-down state for the memory devices. Active background power goes to clocking the PCM's internal row buffers and other peripheral circuitry when the chip is ready to receive commands. Each PCM chip dissipates idle background power when the clock is off, but the chip is still "on." The value here is for a commercially available PCM device [19]. We model the time to "wake up" from this state based on values in [19]. The model does not include power regulators or other on-board overheads.

The baseline Moneta array with four memory controllers, but excluding the memory itself, consumes a maximum of 3.19 W. For 8 KB transfers, the memory controller bandwidth limits memory power consumption to 0.6 W for writes, since transfer time over the DDR pins limits the frequency of array writes. Transfer time and power for reads is the same, but array power is lower. The PCIe link limits total array active memory power to 0.4 W (2 GB/s writes at 16.82 pJ/bit and 2 GB/s reads at 2.47 pJ/bit, plus row buffer power). Idle power for the array is also small – 0.13 W or 2 mW/GB. Total power for random 8 KB writes is 3.47 W, according to our model.

For small writes (e.g., 512 bytes), power consumption for the memory at a single controller could potentially reach 8.34 W since the transfer time is smaller and the controller can issue writes more frequently. For these accesses, peak controller bandwidth is 1.4 GB/s. Total Moneta power in this case could be as high as 11 W. As a result, efficiency for 512 byte writes in terms of MB/s/W drops by 85% compared to 8 KB writes.

The source of this inefficiency is a mismatch between access size and PCM row width. For small requests, the array reads or writes more bits than are necessary to satisfy the requests. Implementing selective writes [10] resolves this problem and limits per-controller power consumption to 0.6 W regardless of access size. Selective writes do not help with reads, however. For reads, peak per-controller bandwidth and power are 2.6 GB/s and 2.5 W, respectively, for 512 byte accesses.

An alternative to partial read and writes is to modify the size of the PCM devices' internal row buffer. Previous work [10] found this parameter to be an important factor in the efficiency of PCM memories meant to replace DRAM. For storage arrays, however, different optimizations are desirable because Moneta's spatial and temporal access patterns are different, and Moneta guarantees write durability.

Moneta's scheduler issues requests that are at least 512 bytes and usually several KB. These requests result in sequential reads and writes to memory that have much higher spatial locality than DRAM accesses that access a cache line at a time. In addition, since accesses often affect an entire row, there are few chances to exploit temporal locality. This means that, to optimize efficiency and performance, the row size and stripe size should be the same and they should be no smaller than the expected transfer size.

Finally, Moneta's durability requirements preclude write coalescing, a useful optimization for PCM-based main memories [10]. Coalescing requires keeping the rows open to opportunistically merge writes, but durability requires closing rows after accesses complete to ensure that the data resides in non-volatile storage rather than the volatile buffers.

VI. Evaluation

This section compares Moneta's performance to other storage technologies using both microbenchmarks and full-fledged applications. We compare Moneta to three other high-performance storage technologies (Table II) – a RAID array of disks, a RAID array of flash-based SSDs, and an 80GB FusionIO flash-based PCIe card. We also estimate performance for Moneta with a 4x PCIe link that matches the peak bandwidth of the other devices.

We tuned the performance of each system using the IO tuning facilities available for Linux, the RAID controller, and the FusionIO card. Each device uses the best-performing of Linux's IO schedulers: For RAID-disk and RAID-SSD the no-op scheduler performs best. The FusionIO card uses a custom driver which bypasses the Linux IO scheduler just as Moneta's does. We use software RAID for the SSD rather than hardware because our measurements and previous studies [4] show better performance for that configuration.

Component	Idle	Peak
Scheduler & DMA [20]	0.3 W	1.3 W
Ring [21]	0.03 W	0.06 W
Scheduler buffers [22]	1.26 mW	4.37 mW
Memory controller [21]	0.24 W	0.34 W
Mem. ctrl. buffers [22]	1.26 mW	4.37 mW
PCIe [20]	0.12 W	0.4 W
PCM write [10]		16.82 pJ/bit
PCM read [10]		2.47 pJ/bit
PCM buffer write [10]		1.02 pJ/bit
PCM buffer read [10]		0.93 pJ/bit
PCM background [10, 23]	264 μW/die	20 μW/bit

Table I. **Moneta power model** The component values for the power model come from datasheets, Cacti [22], [21], and the PCM model in [10]

Name	Bus	Description
RAID-Disk	PCIe 1.1 ×4 1 GB/s	4×1TB hard drives. RAID-0 PCIe controller.
RAID-SSD	SATA II ×4 1.1 GB/s	4×32GB X-25E SSDs. RAID-0 Software RAID.
FusionIO	PCIe 1.1 ×4 1 GB/s	Fusion-IO 80GB PCIe SSD
Moneta-4x	PCIe 1.1 ×4 1 GB/s	See text.
Moneta-8x	PCIe 1.1 ×8 2 GB/s	See text.

Table II. **Storage arrays** We compare Moneta to a variety of existing disk and SSD technologies. Bus bandwidth is the peak bandwidth each device's interface allows.

A. Microbenchmarks

Figure 5 and Table III contain the results of the comparison. For bandwidth, Moneta+Spin outperforms the other arrays by a wide margin, although for large transfer sizes most of this benefit is due to its faster PCIe link. For mixed reads and writes, Moneta delivers between 8.9× (vs. FusionIO) and 616× (vs. disk) more bandwidth. For Moneta+Spin-4x, the gains are smaller (4.4-308×). For mixed 512 KB read and write accesses Moneta is 32× faster than disk and 4.5× faster than FusionIO. The bandwidth gains relative to the SSDs vary less widely – between 7.6 and 89× for Moneta and between 3.8 and 44× for Moneta+Spin-4x.

Moneta also delivers large gains in terms of IOPS. Moneta sustains between 364 and 616× more 4 KB IOPS than disk and between 4 and 8.9× more than FusionIO. The gains for small transfers are much larger: Moneta achieves 1.1M 0.5 KB IOPS, or 10× the value for FusionIO.

Table III also shows the number of instructions each device requires to perform a 4 KB read or write request. Moneta's op-

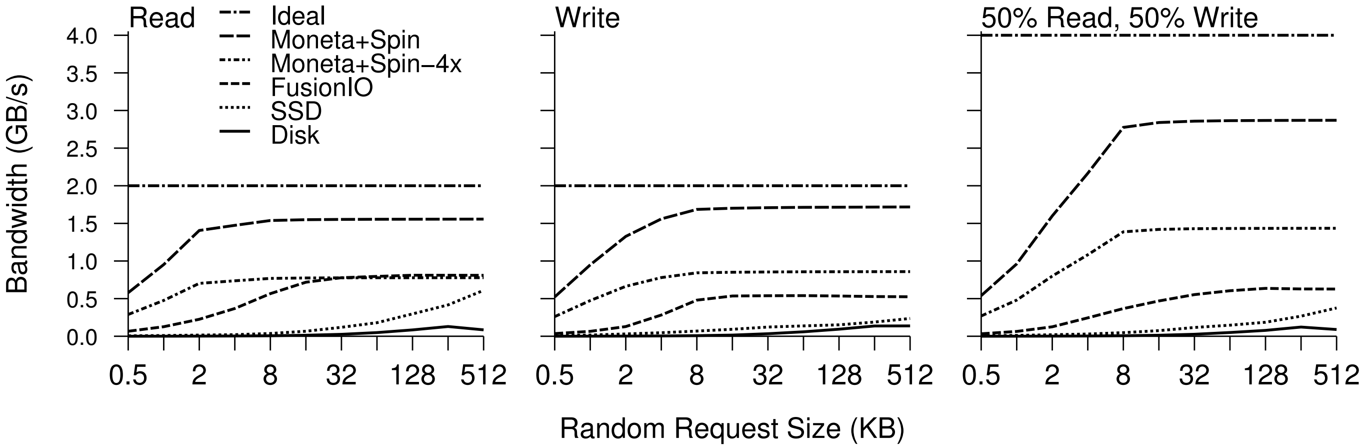

Fig. 5. **Storage array performance comparison** Small request sizes measure random access performance, while larger transfers measure sequential accesses. Moneta+Spin-4x conservatively models a 1GB/s PCIe connection and matches or outperforms the other storage arrays.

	4KB IOPS		512 IOPS		Instructions per 4 KB IOP	
	Read	Write	Read	Write	Read	Write
RAID-Disk	863	1,070	869	1,071	32,042	34,071
RAID-SSD	5,256	11,714	11,135	17,007	31,251	32,540
FusionIO	92,225	70,579	132,826	71,338	49,369	60,147
Moneta+Spin	368,534	389,859	1,156,779	1,043,793	17,364	20,744

Table III. **IO operation performance** Moneta's combination of fast storage and optimized software stack allow it to perform 1.1M 512 byte IOPS and 370K 4 KB IOPS. Software optimizations also reduces the number of instructions per IO operations by between 38 and 47%.

timized driver requires between 38 and 47% fewer instructions than RAID-Disk and RAID-SSD. The reduction relative to the baseline Moneta software stack is similar. As a result, it takes five processors (rather than eleven) to fully utilize the Moneta array. FusionIO's driver requires between 1.4 and 1.8× more instructions than the RAID arrays because it caches device meta-data to increase performance.

B. Applications

Moneta's optimized hardware interface and software stack eliminates most of the operating system overheads that limited the performance for raw device access. Here, we explore how that performance translates to the application level and highlight where further software optimizations may be useful.

Table IV describes the workloads we use in this evaluation. They fall into three categories that represent potential uses for Moneta: IO benchmarks that use typical system calls to access storage, database applications that implement transaction systems with strong durability guarantees, and paging applications that use storage to back virtual memory. Whenever practical and possible, we use direct IO to bypass the system buffer cache for Moneta and FusionIO, since it improves performance. Adding the ability to apply this setting without modifying application source code would be a useful optimization.

Table V shows the performance of the storage arrays and the speedup of Moneta over each array. All data use XFS [24] as the underlying file system because it efficiently supports parallel file access.

The performance of the IO benchmarks improves significantly running on Moneta: On average, Moneta is 9.48× faster than RAID-disk, 2.84× faster than RAID-SSD, and 2.21× faster than FusionIO. However, the performance of XDD-Random on Moneta is much lower when reads and writes go through the file system instead of going directly to the raw block device. This suggests that further optimizations to the file system may provide benefits for Moneta, just as removing device driver overheads did. Performance on Build is especially disappointing, but this is in part because there is no way to disable the file buffer cache for that workload.

The results for database applications also highlight the need for further software optimizations. Databases use complex buffer managers to reduce the cost of IO, but our experience with Moneta to date suggests reevaluating those optimizations. The results for Berkeley DB bear this out: Berkeley DB is simpler than the PostgreSQL and MySQL databases that OLTP and PTFDB use, and that simplicity may contribute to Moneta's larger benefit for Berkeley DB.

For the paging applications, Moneta is on average 7.33×, 1.61×, and 3.01× faster than RAID-Disk, RAID-SSD, and FusionIO, respectively. For paging applications, Moneta is on average 2.75× faster than the other storage arrays. Not surprisingly, the impact of paging to storage depends on the memory requirements of the program, and this explains the

Name	Data footprint	Description
IO benchmarks		
XDD-Sequential-Raw	55 GB	4MB sequential reads/writes from 16 threads through the raw block device
XDD-Sequential-FS	55 GB	4MB sequential reads/writes from 16 threads through the file system
XDD-Random-Raw	55 GB	4KB random reads/writes from 16 threads through the raw block device
XDD-Random-FS	55 GB	4KB random reads/writes from 16 threads through the file system
Postmark	≈0.3-0.5 GB	Models an email server
Build	0.5 GB	Compilation of the Linux 2.6 kernel
Database applications		
PTFDB	50 GB	Palomar Transient Factory database realtime transient sky survey queries
OLTP	8 GB	Transaction processing using Sysbench running on a MySQL database
Berkeley-DB Btree	4 GB	Transactional updates to a B+tree key/value store
Berkeley-DB Hashtable	4 GB	Transactional updates to a hash table key/value store
Paging applications		
BT	11.3 GB	Computational fluid dynamics simulation
IS	34.7 GB	Integer sort with the bucket sort algorithm
LU	9.4 GB	LU matrix decomposition
SP	11.9 GB	Simulated CFD code solves Scalar-Pentadiagonal bands of linear equations
UA	7.6 GB	Solves a heat transfer problem on an unstructured, adaptive grid

Table IV. **Benchmarks and applications** We use a total of fifteen benchmarks and workloads to compare Moneta to other storage technologies.

Workload	Raw Performance				Speedup of Moneta vs.		
	RAID-Disk	RAID-SSD	FusionIO	Moneta	RAID-Disk	RAID-SSD	FusionIO
IO benchmarks							
Postmark	4,080.0 s	46.8 s	36.7 s	27.4 s	149.00×	1.71×	1.34×
Build	80.9 s	36.9 s	37.6 s	36.9 s	2.19×	1.00×	1.02×
XDD-Sequential-Raw	244.0 MB/s	569.0 MB/s	642.0 MB/s	2,932.0 MB/s	12.01×	5.15×	4.57×
XDD-Sequential-FS	203.0 MB/s	564.0 MB/s	641.0 MB/s	2,773.0 MB/s	13.70×	4.92×	4.35×
XDD-Random-Raw	1.8 MB/s	32.0 MB/s	142.0 MB/s	1,753.0 MB/s	973.89×	54.78×	12.35×
XDD-Random-FS	3.3 MB/s	30.0 MB/s	118.0 MB/s	261.0 MB/s	80.40×	8.70×	2.21×
				Harmonic mean	9.48×	2.84×	2.21×
Database applications							
PTFDB	68.1 s	5.8 s	2.3 s	2.1 s	32.60×	2.75×	1.11×
OLTP	304.0 tx/s	338.0 tx/s	665.0 tx/s	799.0 tx/s	2.62×	2.36×	1.20×
Berkeley-DB BTree	253.0 tx/s	6,071.0 tx/s	5,975.0 tx/s	14,884.0 tx/s	58.80×	2.45×	2.49×
Berkeley-DB HashTable	191.0 tx/s	4,355.0 tx/s	6,200.0 tx/s	10,980.0 tx/s	57.50×	2.52×	1.77×
				Harmonic mean	8.95×	2.51×	1.48×
Paging applications							
BT	84.8 MIPS	1,461.0 MIPS	706.0 MIPS	2,785.0 MIPS	32.90×	1.90×	3.94×
IS	176.0	252.0	252.0	351.0	2.00×	1.39×	1.39×
LU	59.1	1,864.0	1,093.0	4,298.0	72.70×	2.31×	3.93×
SP	87.0	345.0	225.0	704.0	8.08×	2.04×	3.13×
UA	57.2	3,775.0	445.0	3,992.0	69.80×	1.06×	8.97×
				Harmonic mean	7.33×	1.61×	3.01×

Table V. **Workload performance** We run our fifteen benchmarks and applications on the RAID-Disk, RAID-SSD, FusionIO, and Moneta storage arrays, and we show the speedup of Moneta over the other devices.

large variation across the five applications. Comparing the performance of paging as opposed to running these applications directly in DRAM shows that Moneta reduces performance by only 5.90× as opposed to 14.7× for FusionIO, 13.3× for RAID-SSD, and 83.0× for RAID-Disk. If additional hardware and software optimizations could reduces this overhead, Moneta-like storage devices could be come a viable option for increasing effective memory capacity.

VII. Related work

Software optimizations and scheduling techniques for IO operations for disks and other technologies has been the subject of intensive research for many years [25–31]. The main challenge in disk scheduling is minimizing the impact of the long rotational and seek time delays. Since Moneta does not suffer from these delays, the scheduling problem focuses on exploiting parallelism within the array and minimizing hardware and software overheads.

Recently, scheduling for flash-based solid-state drives has received a great deal of attention [32–34]. These schedulers focus on reducing write overheads, software overheads, and exploiting parallelism within the drive. The work in [35] explores similar driver optimizations for a PCIe-attached, flash-based SSD (although they do not modify the hardware interface) and finds that carefully-tuned software scheduling is useful in that context. Our results found that any additional software overheads hurt Moneta's performance, and that scheduling in hardware is more profitable.

In the last decade, there have also been several proposals for software scheduling policies for MEMS-based storage arrays [36–40]. Other researchers have characterized SSDs [41–43] and evaluated their usefulness on a range of applications [44–47]. Our results provide a first step in this direction for faster non-volatile memories.

Researchers have developed a range of emulation, simulation, and prototyping infrastructures for storage systems. Most of these that target disks are software-based [41, 48–51]. Recently, several groups have built hardware emulation systems for exploring flash-based SSD designs: The work in [52–54] implements flash memory controllers in one or more FPGAs and attach them to real flash devices. Moneta provides a similar capability for fast non-volatile memories, but it emulates the memories using the memory controller we described in Section II. The work in [52] uses the same BEE3 platform that Moneta uses.

VIII. Conclusion

We have presented Moneta, a storage array architecture for advanced non-volatile memories. We emulate Moneta using FPGAs and modified memory controllers that can model PCM memory using DRAM. A series of software and hardware interface optimizations significantly improves Moneta's performance. Our exploration of Moneta designs shows that optimizing PCM for storage applications requires a different set of trade-offs than optimizing it as a main memory replacement. In particular, memory array latency is less critical for storage applications if sufficient parallelism is available, and durability requirements prevent some optimizations.

Optimizations to Moneta's hardware and software reduce software overheads by 62% for 4 KB operations, and enable sustained performance of 1.1M 512-byte IOPS and 541K 4 KB IOPS with a maximum sustained bandwidth of 2.8 GB/s. Moneta's optimized IO stack completes a single 512-byte IOP in 9 μs. Moneta speeds up a range of file system, paging, and database workloads by up to 8.7× compared to a state-of-the-art flash-based SSD with harmonic mean of 2.1×, while consuming a maximum power of 3.2 W.

References

[1] "Wd velociraptor: Sata hard drives," http://www.wdc.com/wdproducts/library/SpecSheet/ENG/2879-701284.pdf.

[2] "Ibm 3340 disk storage unit," http://www-03.ibm.com/ibm/history/exhibits/storage/storage_3340.html.

[3] "Intel x58 express chipset product brief," 2008, http://www.intel.com/products/desktop/chipsets/x58/x58-overview.htm.

[4] J. He, A. Jagatheesan, S. Gupta, J. Bennett, and A. Snavely, "Dash: a recipe for a flash-based data intensive supercomputer," New York, NY, USA, 2010.

[5] B. Dieny, R. Sousa, G. Prenat, and U. Ebels, "Spin-dependent phenomena and their implementation in spintronic devices," *VLSI Technology, Systems and Applications, 2008. VLSI-TSA 2008. International Symposium on*, pp. 70–71, April 2008. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=04530803

[6] Y. Ho, G. Huang, and P. Li, "Nonvolatile memristor memory: Device characteristics and design implications," in *Computer-Aided Design - Digest of Technical Papers, 2009. ICCAD 2009. IEEE/ACM International Conference on*, 2-5 2009, pp. 485 –490.

[7] S. Parkin, "Racetrack memory: A storage class memory based on current controlled magnetic domain wall motion," in *Device Research Conference, 2009. DRC 2009*, 22-24 2009, pp. 3 –6.

[8] T. Rueckes, K. Kim, E. Joselevich, G. Y. Tseng, C.-L. Cheung, and C. M. Lieber, "Carbon Nanotube-Based Nonvolatile Random Access Memory for Molecular Computing," *Science*, vol. 289, no. 5476, pp. 94–97, 2000. [Online]. Available: http://www.sciencemag.org/cgi/content/abstract/289/5476/94

[9] M. J. Breitwisch, "Phase change memory," *Interconnect Technology Conference, 2008. IITC 2008. International*, pp. 219–221, June 2008. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=04546972

[10] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," in *ISCA '09: Proceedings of the 36th annual international symposium on Computer architecture*. New York, NY, USA: ACM, 2009, pp. 2–13.

[11] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," *International Symposium on Computer Architecture*, June 2009. [Online]. Available: http://users.ece.utexas.edu/~qk/papers/pcm.pdf

[12] G. Dhiman, R. Ayoub, and T. Rosing, "Pdram: a hybrid pram and dram main memory system," in *DAC '09: Proceedings of the 46th Annual Design Automation Conference*. New York, NY, USA: ACM, 2009, pp. 664–469.

[13] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A durable and energy efficient main memory using phase change memory technology," in *ISCA '09: Proceedings of the 36th annual international symposium on Computer architecture*. New York, NY, USA: ACM, 2009, pp. 14–23.

[14] S. Cho and H. Lee, "Flip-n-write: a simple deterministic technique to improve pram write performance, energy and endurance," in *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. New York, NY, USA: ACM, 2009, pp. 347–357.

[15] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali, "Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling," in *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. New York, NY, USA: ACM, 2009, pp. 14–23.

[16] A. Huffman and J. Clark, "Serial ata native command queuing."

[17] "The ramp project," http://ramp.eecs.berkeley.edu/index.php?index.

[18] "Xdd version 6.5," http://www.ioperformance.com/.

[19] Elpida, "Elpida ddr2 sdram ede1104afse datasheet," 2008, http://www.elpida.com/pdfs/E1390E30.pdf.

[20] Intel, "Intel system controller hub datasheet," April 2008, http://download.intel.com/design/chipsets/embedded/datashts/319537.pdf.

[21] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures," in *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009, pp. 469–480.

[22] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi, "Cacti 5.1," HP Labs, Palo Alto, Tech. Rep. HPL-2008-20, 2008. [Online]. Available: http://www.hpl.hp.com/techreports/2008/HPL-2008-20.html

[23] Numonyx.

[24] S. G. International, "Xfs: A high-performance journaling filesystem," http://oss.sgi.com/projects/xfs.

[25] Y. J. Nam and C. Park, "Design and evaluation of an efficient proportional-share disk scheduling algorithm," *Future Gener. Comput. Syst.*, vol. 22, no. 5, pp. 601–610, 2006.

[26] A. Thomasian and C. Liu, "Disk scheduling policies with lookahead," *SIGMETRICS Perform. Eval. Rev.*, vol. 30, no. 2, pp. 31–40, 2002.

[27] B. L. Worthington, G. R. Ganger, and Y. N. Patt, "Scheduling algorithms for modern disk drives," in *SIGMETRICS '94: Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and modeling of computer systems*. New York, NY, USA: ACM, 1994, pp. 241–251.

[28] M. Seltzer, P. Chen, and J. Ousterhout, "Disk scheduling revisited," in *Proceedings of the 1990 Winter Usenix*, 1990, pp. 313–324.

[29] T. J. Teorey and T. B. Pinkerton, "A comparative analysis of disk scheduling policies," *Commun. ACM*, vol. 15, no. 3, pp. 177–184, 1972.

[30] P. J. Shenoy and H. M. Vin, "Cello: a disk scheduling framework for next generation operating systems," *SIGMETRICS Perform. Eval. Rev.*, vol. 26, no. 1, pp. 44–55, 1998.

[31] S. W. Ng, "Improving disk performance via latency reduction," *IEEE Trans. Comput.*, vol. 40, no. 1, pp. 22–30, 1991.

[32] J. Kim, Y. Oh, E. Kim, J. Choi, D. Lee, and S. H. Noh, "Disk schedulers for solid state drivers," in *EMSOFT '09: Proceedings of the seventh ACM international conference on Embedded software*. New York, NY, USA: ACM, 2009, pp. 295–304.

[33] M. Dunn and A. L. N. Reddy, "A new i/o scheduler for solid state devices," Department of Electrical and Computer Engineering Texas A&M University, Tech. Rep. TAMU-ECE-2009-02-3, 2009.

[34] Y.-B. Chang and L.-P. Chang, "A self-balancing striping scheme for nand-flash storage systems," in *SAC '08: Proceedings of the 2008 ACM symposium on Applied computing*. New York, NY, USA: ACM, 2008, pp. 1715–1719.

[35] E. Seppanen, M. T. OKeefe, and D. J. Lilja, "High performance solid state storage under linux," in *Proceedings of the 30th IEEE Symposium on Mass Storage Systems*, 2010.

[36] E. Lee, K. Koh, H. Choi, and H. Bahn, "Comparison of i/o scheduling algorithms for high parallelism mems-based storage devices," in *SEPADS'09: Proceedings of the 8th WSEAS International Conference on Software engineering, parallel and distributed systems*. Stevens Point, Wisconsin, USA: World Scientific and Engineering Academy and Society (WSEAS), 2009, pp. 150–155.

[37] H. Bahn, S. Lee, and S. H. Noh, "P/pa-sptf: Parallelism-aware request scheduling algorithms for mems-based storage devices," *Trans. Storage*, vol. 5, no. 1, pp. 1–17, 2009.

[38] S. W. Schlosser, J. L. Griffin, D. F. Nagle, and G. R. Ganger, "Designing computer systems with mems-based storage," *SIGOPS Oper. Syst. Rev.*, vol. 34, no. 5, pp. 1–12, 2000.

[39] M. Uysal, A. Merchant, and G. A. Alvarez, "Using mems-based storage in disk arrays," in *FAST '03: Proceedings of the 2nd USENIX Conference on File and Storage Technologies*. Berkeley, CA, USA: USENIX Association, 2003, pp. 89–101.

[40] I. Dramaliev and T. Madhyastha, "Optimizing probe-based storage," in *FAST '03: Proceedings of the 2nd USENIX Conference on File and Storage Technologies*. Berkeley, CA, USA: USENIX Association, 2003, pp. 103–114.

[41] K. El Maghraoui, G. Kandiraju, J. Jann, and P. Pattnaik, "Modeling and simulating flash based solid-state disks for operating systems," in *WOSP/SIPEW '10: Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering*. New York, NY, USA: ACM, 2010, pp. 15–26.

[42] C. Dirik and B. Jacob, "The performance of pc solid-state disks (ssds) as a function of bandwidth, concurrency, device architecture, and system organization," in *ISCA '09: Proceedings of the 36th annual international symposium on Computer architecture*. New York, NY, USA: ACM, 2009, pp. 279–289.

[43] F. Chen, D. A. Koufaty, and X. Zhang, "Understanding intrinsic characteristics and system implications of flash memory based solid state drives," in *SIGMETRICS '09: Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*. New York, NY, USA: ACM, 2009, pp. 181–192.

[44] D. Narayanan, E. Thereska, A. Donnelly, S. Elnikety, and A. Rowstron, "Migrating server storage to ssds: analysis of tradeoffs," in *EuroSys '09: Proceedings of the 4th ACM European conference on Computer systems*. New York, NY, USA: ACM, 2009, pp. 145–158.

[45] S.-W. Lee, B. Moon, C. Park, J.-M. Kim, and S.-W. Kim, "A case for flash memory ssd in enterprise database applications," in *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 2008, pp. 1075–1086.

[46] S. Chen, "Flashlogging: exploiting flash devices for synchronous logging performance," in *SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 2009, pp. 73–86.

[47] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan, "Fawn: a fast array of wimpy nodes," in *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. New York, NY, USA: ACM, 2009, pp. 1–14.

[48] G. Ganger, B. Worthington, and Y. Patt, "Disksim," http://www.pdl.cmu.edu/DiskSim/.

[49] C. Ruemmler and J. Wilkes, "An introduction to disk drive modeling," *Computer*, vol. 27, no. 3, pp. 17–28, 1994.

[50] J. L. Griffin, J. Schindler, S. W. Schlosser, J. C. Bucy, and G. R. Ganger, "Timing-accurate storage emulation," in *FAST '02: Proceedings of the 1st USENIX Conference on File and Storage Technologies*. Berkeley, CA, USA: USENIX Association, 2002, p. 6.

[51] D. Kotz, S. B. Toh, and S. Radhakrishnan, "A detailed simulation model of the hp 97560 disk drive," Dartmouth College, Hanover, NH, USA, Tech. Rep., 1994.

[52] J. D. Davis and L. Zhang, "Frp: A nonvolatile memory research platform targeting nand flash," in *Proceedings of First Workshop on Integrating Solid-State Memory in the Storage Hierarchy*, 2009.

[53] S. Lee, K. Fleming, J. Park, K. Ha, A. M. Caulfield, S. Swanson, Arvind, and J. Kim, "Bluessd: An open platform for cross-layer experiments for nand flash-based ssds," in *Proceedings of The 5th Workshop on Architectural Research Prototyping*, 2010.

[54] L. M. Grupp, A. M. Caulfield, J. Coburn, J. Davis, and S. Swanson, "Beyond the datasheet: Using test beds to probe non-volatile memories' dark secrets," in *To appear: IEEE Globecom 2010 Workshop on Application of Communication Theory to Emerging Memory Technologies (ACTEMT 2010)*, 2010.